University of Toronto
CSC2108F - Automated Verification

A Case Study Production Cell with PROMELA/SPIN

Dimitrie O. Păun
dimi@cs.toronto.edu

Bernd Biechele
biechele@cs.toronto.edu

Abstract

In this project we develop a software controller for a production cell system. The main objective is to find out if PROMELA/SPIN is an appropriate environment for developing embedded real-time systems. We describe the production cell hardware model and its requirements. Subsequently, we describe the model of a distributed and concurrent controller for it. Furthermore, we mention some required properties we could prove with SPIN based on our model as well as those we could not prove. We outline a transformation scheme of the PROMELA language to C code. Finally, we describe some experiences and draw conclusions.

Contents

1 Introduction

The course CSC2108 "Automated Verification" provides an overview of formal methods, highlighting some, e.g. SMV, PROMELA/SPIN, Concurrency Workbench and PVS. Although it has been shown on many examples that these methods are useful and help to develop correct software, they are still not widely used in industry.

One approach to support development of correct software is model checking. A model checker starts from some sort of specification and tries to explore all possible states that an execution of this specification might have. SPIN[1], as described in [HO 97] is one of such tools, developed at Bell Labs. PROMELA, its input language, is a non-deterministic language, loosely based on Dijkstra's guarded command notation and Hoare's language CSP. It allows to specify synchronous and asynchronous message passing via channels.

The case study "Control Software for an Industrial Production Cell"[2], as described in [LL 95], was initiated at the Forschungszentrum Informatik (FZI) in Karlsruhe, Germany. Its intention was to show the usefulness of formal methods for critical software systems and to show their applicability to real-world examples. Many authors contributed to this case study and 18 of these contributions are summarized and compared in [LL 95].

2 The Task

The goal of the project is to model, design and implement a production cell controller that has certain proven properties. The production cell processes metal blanks which are conveyed to a press by a feed belt. A robot takes each blank from the feed belt and places it into the press. The robot arm withdraws from the press, the press processes the metal blank and opens again. Finally, the robot takes the forged metal plate out of the press and puts it on a deposit belt.

Figure 1 shows the physical configuration of the cell. The general sequence for processing a metal plate is the following:

1. The feed belt conveys the metal plate to the elevating rotary table.
2. The elevating rotary table is moved to a position adequate for unloading by the first robot arm.

[1] http://netlib.bell-labs.com/netlib/spin/whatispin.html
[2] http://www.fzi.de/prost/projects/production_cell/ProductionCell.html

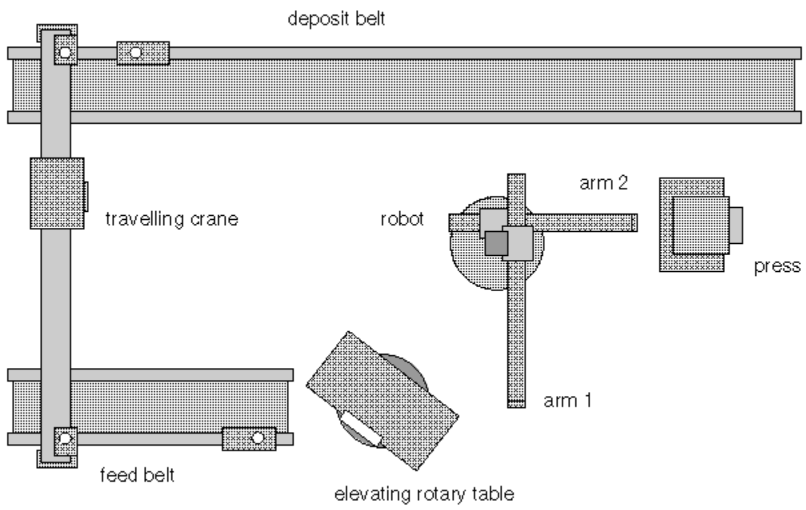

Figure 1: Top view of the production cell

3. The first robot arm picks up the metal plate.
4. The robot rotates counterclockwise so that arm 1 points to the open press, places the metal plate into it and then withdraws from the press.
5. The press forges the metal blank and opens again.
6. The robot retrieves the metal plate with its second arm, rotates further and unloads the plate on the deposit belt.
7. The deposit belt transports the plate to the traveling crane.
8. The traveling crane picks up the metal plate, moves to the feed belt, and unloads the metal plate on it.

Of course, this is a rather simplified description of the system. For the complete specification of the production cell, refer to [LL 95].

3 Modeling Reactive Systems

This section describes the way we modeled the system. We explain some of the abstractions, modeling decisions and drawbacks of our approach.

3.1 Overview

Our model was strongly influenced by the need to minimize the state space requirements for checking the model. Since SPIN does not provide any means of modeling time, we abstract from the reactive components by separating them from the rest of the system. Clearly, such a simplification might not be valid in the general case. However, we argue below that under certain assumptions this simplification is valid.

Consider the process in Figure 2. Clearly, if the reactive component does

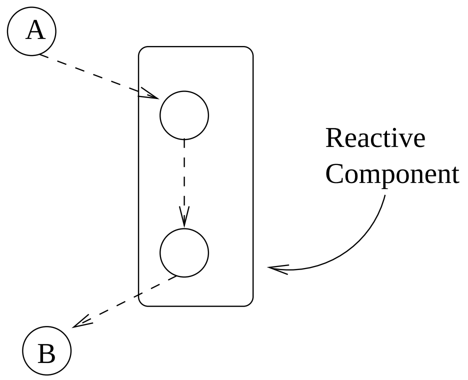

Figure 2: Process with a serial reactive component

not depend on the state of the environment and if it terminates in a finite amount of time, this process is equivalent to the one in Figure 3 as far as SPIN is concerned, since in SPIN a transition may take any finite amount of time.

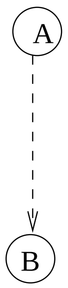

Figure 3: Equivalent process without the reactive component

We do not attempt to model and/or verify any of the reactive subsystems. All subsystems we use fall in the category of feedback loop systems and have the form:

```
start(actuator)
wait_until(sensor <op> value)
stop(actuator)
```

where the specifications are that the `stop(actuator)` action is performed no later than a given time delay after the condition `sensor <op> value` becomes true. The assumption that a reactive component completes a task in a finite amount of time is equivalent in the vast majority of cases to the assumption that the component will not fail.

That each reactive component takes a finite amount of time to finish has to be validated manually though.

3.2 Disadvantages

The preconditions required for this abstraction to work are:

1. The reactive components must be executed sequentially. That is, the process that uses the reactive component must block until the component finishes its job (Figure 2 tries to capture this requirement). Note that this is a requirement on how the reactive component is *used* and not how it is *implemented*.

2. The reactive components should not depend on the state of any other process.

These preconditions may be argued (and rightly so) to be too strong. Clearly, obeying these rules would most likely lead to a less then optimal implementation. In certain cases, the performance hit may be unacceptable.[3]

A more serious disadvantage is that the resulting model can not be used directly by the customer to verify the correctness of the implementation.

[3] For example, consider a magnetic head that can move along both axes across a square table. To move from one corner of the table to the diagonally opposed one, the head can first move along one axis and, at the end of the run, move across the other axis (that is, it can do the two movements serially) or it can move on the diagonal by moving across both axes simultaneous (that is, it can perform the two movements in parallel). If the head is controlled by only one process, the first precondition means that the system has to to perform the two movements serially which, in many cases, may not be acceptable.

The model can only ensure the programmer the implementation is correct. [4]

3.3 Justification

Why follow this path? From our previous experience with SPIN, we knew that it is very easy to exceed the memory constraints which, at the moment of writing, are about 128MB of main memory. We wanted to actually implement a system that can be formally verified. We did not want to check just a high-level abstraction of the system, but rather a detailed enough implementation that would allow us an almost mechanical translation to C. We decided that an approach that will allow us to achieve this goal even at the expense of provability of the correctness to the customer was an acceptable approximation. Moreover, through careful coding techniques, we think that we are able to at least convince (if not to prove to) the customer that the system indeed does what it is supposed to do.

As for the performance loss due to the independence in the reactive subsystems, we decided that in this particular case the performance penalty is indeed small because, as we will show in Section 4, most parallelism in the system is still maintained.

3.4 An Alternative

A better model of a reactive system would include processes that model the sensors and channels for the actuators. This way, the sensors and the actuator channels can be connected to processes that are provided by the customer to model the environment. Such a model can then be used to prove to a customer that the system does indeed function according to the specifications in the given environment. Unfortunately, due to time constraints, we did not expand the model in this direction.

4 The Design of The Controller

This section describes our approach to the design of the production cell controller software. We explain design decisions, coding techniques and properties we were able to prove.

[4]We use the term *customer* for the party that provides the specification and the term *programmer* for the party that provides the implementation of that specification.

4.1 Overview

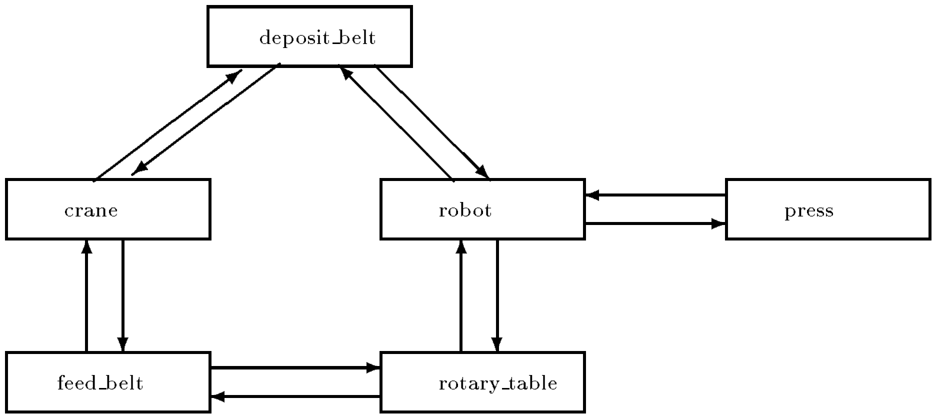

Figure 4: Components of the PROMELA model

We opted for a distributed and object-oriented design. Each component is controlled by a process which only has knowledge about its immediate neighbours. A process has an exclusive access to the component's sensors and actuators. This distributed design allows for a high degree of parallelism between the processes while keeping the design simple.

The architecture of the design is depicted in Figure 4. Each box represents a component and is implemented as a PROMELA `proctype`, i.e. as a process. All these processes run concurrently; just one instance of each process is created. The processes exchange information through directed communication channels and global variables. The exchange of information is indicated by arrows in the figure. The channel communications are synchronous, i.e. the sender (respectively, the receiver) blocks until the receiver is ready to receive (respectively, the sender is ready to send).

As mentioned before, we abstracted from the hardware. To prove some properties, however, we needed to know the state of some of the hardware. One way to achieve this is to set some global variables each time the state of a hardware component changes, since variables used in proofs must be global.

However, directly setting variables to reflect the state of the hardware makes the code harder to read and maintain as the semantics of the action is lost. Moreover, the state space is enlarged unnecessarily since only a small subset of these variables is needed to prove a single property.

A solution to this problem is to use macros of the form:

```
ROBOT_MOVE(ARM1_PRESS)
```

Such macros solve both problems mentioned above since they preserve the semantics and can be defined conditionally to expand to the appropriate variable assignment or an empty string if that variable is not needed for the desired test.

A small problem still remains. How does one know that one verifies the same model? That is, since the macros conditionally expand to different code fragments, how can one ensure that these expansions do not alter the model? Here is one possible method:

- Since the variables are declared only when needed, one can simply compile the source with all the macros expanded to nothing and the variables undeclared. If the code uses the variables in any way, the compiler should generate an error. Alternatively, one can search the source code for variable names. They should only appear in the `never` claim and the macros definitions.

- The macros should only *assign* values to the variable; they should not generate code that tests those variables in any way. This is easy to check by looking at the definition of the macro. Note that the use of macros greatly helps here since one has to look only at the macro definition instead of checking all occurrences of the variable in the code.

4.2 Proving Properties

The properties listed in the production cell specification can be divided into two sorts: properties involving timed behaviour and properties that are independent of time.

4.3 Timed Properties

Properties in this category refer to reactive behaviour. Since we abstracted from the reactive components, we were not able to prove these properties. All these properties prescribe behaviour that can be reduced to the following type of specification:

The system should react no later than t seconds from the moment sensor s changes state (or reaches value v) by turning on/off actuator a.

Since we can not represent time in our model, these restrictions can not be checked or guaranteed.

4.4 Untimed Properties

This class of properties can be further subdivided into *safety* ("Nothing bad happens") and *liveness* ("Something good will eventually happen"). We present one property from each category. The rest of the properties appear in the Appendix A. The properties are expressed in linear time logic (LTL):

A Liveness Property *Every blank introduced into the system via the feed belt will eventually be dropped by the crane on the feed belt again and will have been forged.*

$$\Box((\mathit{forged}[X] \Rightarrow \Diamond\neg\mathit{forged}[X]) \wedge (\neg\mathit{forged}[X] \Rightarrow \Diamond\mathit{forged}[X]))$$

We introduced an array `forged` that keeps track of the state of each blank in the system. Initially, each blank is unforged, i.e. its value in the array is `false`. After being forged in the press, the value becomes `true`. When the traveling crane puts the blank back onto the feed belt, the blank is marked unforged again.

A Safety Property *The magnet of arm 1 may only be deactivated if the arm points towards the press and the arm is extended such that it reaches the press.*

$$\begin{aligned}&\Box((\mathit{robot\_mgn}[0] \wedge \circ\neg\mathit{robot\_mgn}[0]) \Rightarrow\\ &\quad(\mathit{robot\_pos} = \mathit{POS\_ARM1\_PRESS} \wedge\\ &\qquad\mathit{robot\_ext}[0] = \mathit{EXT\_INPRESS}))\end{aligned}$$

If the magnet on arm 1 is deactivated (i.e. the variable `robot_mgn[0]` changes its value from `true` to `false`) then arm 1 must be positioned inside the press.

The system is designed so that up to 8 blanks can stay in the production cell concurrently. Due to memory constraints, we were able to prove all properties with a full state-space search for up to 3 blanks in the system.

5 The Implementation

On of our goals was to be able to easily translate the PROMELA code into C. That is, we wanted to make the leap from the model to the actual running implementation as small as is reasonably possible. We were helped a lot by the C-like nature of the PROMELA language. For an outline of the translation patterns we used, see Appendix B.

6 Related Work

After finishing most parts of the project, we discovered that similar work on a production cell controller had been done by Thierry Cattel as described in [CA 95]. He chose an approach of several layers with decreasing levels of abstraction.

As described in [CA 95], all properties have been verified on a very high level of abstraction. On the more detailed levels not all properties have been verified. The correspondence between models on different levels of abstraction were validated using separate models in Concurrency Workbench [CPS 93]. Since this modeling was done manually, it is a potential source of errors.

In our approach we verified all properties on a very low level of abstraction. This translates in a high level of confidence in the resulting code considering the small distance between the PROMELA model and the actual code.

Our approaches differ in that we aimed at deriving a C implementation rather than an implementation in languages that have sophisticated communication primitives, like the one used in [CA 95].

7 Conclusions

The behavior of concurrent processes is too complex to handle manually, even for a fairly simple system. During the development process, the tools detected errors in our model and were of invaluable help in tracking down the problem.

The amount of time spent on modeling and proving properties about the model has been reasonable (about two weeks). The capability of translating LTL formulae to PROMELA automatically was of great help, and it allows the use of a very expressive and powerful language for checking properties.

We think that the project demonstrated that formal methods are usable. It is important to notice, however, that the size of our model is at the limit of what can be done today with SPIN. It is very easy to reach this limit, and due to the nature of the problem, it is not likely that the situation will improve significantly in the future. More complex systems will certainly require a higher level of abstraction than the one used in this project. Thus, a refinement formalism is required if formal methods are to be used for more significant projects.

A The Properties

This section describes state variables and the LTL formulas used to prove the required properties as specified in [LL 95] (with corresponding numbering).

A.1 The State Variables

To prove a property, all variables included in its corresponding formula have to be declared globally. Therefore we introduced state variables that make the state of the processes (which is otherwise local to each `proctype`) visible.

robot_pos Describes the position of the robot. *robot_pos* can take one of the following values:

- ***POS_ARM1_TABLE*** Arm 1 is pointing to the rotary table;
- ***POS_ARM2_PRESS*** Arm 2 is pointing to the press;
- ***POS_ARM2_BELT*** Arm 2 is pointing to the deposit belt;
- ***POS_ARM1_PRESS*** Arm 1 is pointing to the press;
- ***POS_MOVING*** The robot is rotating.

robot_ext[X] Describes the extent of arm X. *robot_ext[X]* can take one of the following values:

- ***EXT_RETRACTED*** The arm is retracted as far as possible;
- ***EXT_EXTENDING*** The arm is extending. Its exact position is unknown;
- ***EXT_RETRACTING*** The arm is retracting. Its exact position is unknown;
- ***EXT_INPRESS*** The arm is extended such that it is in the press;
- ***EXT_ABOVEBELT*** The arm is extended such that is is above the deposit belt. This setting makes sense only for arm 2;
- ***EXT_ABOVETABLE*** The arm is extended such that is is above the rotary table. This setting makes sense only for arm 1.

robot_mgn[X] If true, the magnet of arm X is activated. Otherwise, the magnet is inactive.

head_hor Describes the horizontal position of the crane. *head_hor* can take one of the following values:

HEAD_DEPOSIT The crane is above the deposit belt;

HEAD_FEED The crane is above the feed belt;

HEAD_HORMV The crane is moving between the feed belt and the deposit belt.

head_ver Describes the vertical position of the crane gripper. *head_ver* can take one of the following values:

HEAD_UP The crane gripper is retracted as far as possible;

HEAD_DOWN The crane gripper is extended as far as possible;

HEAD_VERMV The crane gripper is extending/retracting.

head_mgn If true, the crane magnet is activated;

deposit_eng If true, the engine that advances the deposit belt is on (i.e. the deposit belt is advancing), otherwise, the engine is off and the deposit belt is stopped.

deposit_sen If true, sensor 13 is on (that is, an object is sensed at the end of the deposit belt), otherwise sensor 13 is off.

feed_sen If true, sensor 14 is on (that is, an object is sensed at the end of the deposit belt), otherwise sensor 14 is off.

rotary_loading If true, the rotary table is in the correct loading position and is ready to receive blanks. In all other cases, this variable is false.

rotary_unloading If true, the rotary table is in the correct unloading position and it has a blank to unload. In all other cases, this variable is false.

press_closed Is true if the press is closed or it is in the process of closing, false otherwise;

new_ins If true, a new blank is ready to be inserted into the system. The blanks are inserted through the feed belt.

forged[X] If true, blank X has been forged.

A.2 The Formulae

We give a textual definition and a corresponding formula in LTL for each property we were able to prove. The formulae make use of the state variables described above. Note that the formulae make use of $\circ$, the `next` operator.

Liveness Every blank introduced into the system via the feed belt will eventually be dropped by the crane on the feed belt again and will have been forged.

$$\Box((\mathit{forged[X]} \Rightarrow \Diamond\neg\mathit{forged[X]}) \wedge (\neg\mathit{forged[X]} \Rightarrow \Diamond\mathit{forged[X]}))$$

The formula guarantees that blank X will get forged and unforged over and over again. The only place a blank gets forged is in the press and the only place where it gets unforged is when it is dropped by the crane on the feed belt. The checks were repeated with all possible values for X (ie. X=0,1,2,3,...).

2(a) The press may only close when no robot arm is positioned inside it.

$$\begin{array}{l}\Box(\mathit{press\_closed} \Rightarrow \\ \quad ((\mathit{arm1\_retracted} \wedge \mathit{arm2\_retracted}) \vee \\ \quad (\mathit{robot\_pos} = \mathit{POS\_ARM1\_TABLE} \wedge \mathit{arm2\_retracted}) \vee \\ \quad (\mathit{robot\_pos} = \mathit{POS\_ARM2\_BELT} \wedge \mathit{arm1\_retracted})))\end{array}$$

where

$$\begin{array}{l}\mathit{arm1\_retracted} = \mathit{robot\_ext}[0] = \mathit{EXT\_RETRACTED} \\ \mathit{arm2\_retracted} = \mathit{robot\_ext}[1] = \mathit{EXT\_RETRACTED}\end{array}$$

The formula describes a stronger property: the press may close only when both arms are retracted or one of them is pointing away from the press and the other one is retracted.

2(b) A robot arm may only rotate in the proximity of the press if the arm is retracted or if the press is in its upper or lower position.

$$\begin{array}{l}\Box(\mathit{robot\_pos} = \mathit{POS\_MOVING} \Rightarrow \\ \quad (\mathit{robot\_ext[0]} = \mathit{EXT\_RETRACTED} \wedge \\ \qquad \mathit{robot\_ext[1]} = \mathit{EXT\_RETRACTED}))\end{array}$$

We prove a slightly stronger formula: the robot moves only with both arms retracted.

2(c) The traveling crane is not allowed to knock against a belt laterally.

$$\Box(\mathit{head\_hor} = \mathit{HEAD\_HORMV} \Rightarrow \mathit{head\_ver} = \mathit{HEAD\_UP})$$

Again, we prove a stronger statement: the crane moves horizontally only if the head is up. Thus, no lateral collision can occur.

3(a) The magnet of arm 1 may only be deactivated, if the arm points towards the press and the arm is extended such that it reaches the press.

$$\begin{aligned}&\Box((\mathit{robot\_mgn[0]} \wedge \circ\neg \mathit{robot\_mgn[0]}) \Rightarrow\\&\quad(\mathit{robot\_pos} = \mathit{POS\_ARM1\_PRESS} \wedge\\&\qquad \mathit{robot\_ext[0]} = \mathit{EXT\_INPRESS}))\end{aligned}$$

If the magnet on arm 1 is deactivated (ie. dropping a blank), then arm 1 must be positioned inside the press.

3(b) The magnet of arm 2 may only be deactivated, if its position is above the deposit belt.

$$\begin{aligned}&\Box((\mathit{robot\_mgn[1]} \wedge \circ\neg \mathit{robot\_mgn[1]}) \Rightarrow\\&\quad(\mathit{robot\_pos} = \mathit{POS\_ARM2\_BELT} \wedge\\&\qquad \mathit{robot\_ext[1]} = \mathit{EXT\_ABOVEBELT}))\end{aligned}$$

This property is very similar to the previous one, but for the second arm.

3(c) The magnet of the crane may only be deactivated, if its magnet is above the feed belt and sufficiently close to it.

$$\begin{aligned}&\Box((\mathit{head\_mgn} \wedge \circ\neg \mathit{head\_mgn}) \Rightarrow\\&\quad(\mathit{head\_ver} = \mathit{HEAD\_DOWN} \wedge \mathit{head\_hor} = \mathit{HEAD\_FEED}))\end{aligned}$$

If we hold an object and we deactivate the crane magnet, then we must be above the feed belt and with the crane head lowered.

3(d) The feed belt may only convey a blank through its light barrier, if the table is in loading position.

$$\Box((\mathit{feed\_sen} \wedge \circ\neg \mathit{feed\_sen}) \Rightarrow \mathit{rotary\_loading})$$

If a blank leaves the feed belt, then the rotary table must be in the loading position and ready to receive the blank.

3(e) The deposit belt must be stopped after a blank has passed the light barrier at its end and may only be started after the crane has picked up the blank.

$$\Box(((\mathit{deposit\_sen} \wedge \circ\neg\mathit{deposit\_sen}) \Rightarrow \circ\neg(\mathit{deposit\_eng}\,\mathcal{U}\,\mathit{deposit\_sen}))\wedge \\ ((\mathit{deposit\_eng} \wedge \circ\neg\mathit{deposit\_eng}) \Rightarrow \circ(\neg\mathit{deposit\_eng}\,\mathcal{U}\,\mathit{pickup})))$$

where

$$\mathit{pickup} = (\mathit{head\_hor} = \mathit{HEAD\_DEPOSIT} \wedge \\ \mathit{head\_ver} = \mathit{HEAD\_DOWN} \wedge \mathit{head\_mgn})$$

We must prove two properties: that we stop the deposit belt after the blank passes through the light barrier and that we will not start the belt before the crane picks the object up. So, the first conjunct formalizes the requirement that once the object passes through the light barrier (that is, *deposit_sen* goes from true to false), then the engine will stop before the next object reaches the sensor. The second conjunct captures the requirement that once the engine stops, it does not start again until the crane picks up the object.

4(a) A new blank may only be put on the feed belt, if sensor 14 confirms that the last one has arrived at the end of the feed belt.

$$\Box(\mathit{new\_on\_feed} \Rightarrow \circ(\neg\mathit{new\_on\_feed}\,\mathcal{U}\,(\neg\mathit{feed\_sen} \wedge \circ\,\mathit{feed\_sen})))$$

where

$$\mathit{new\_on\_feed} = (\mathit{head\_mgn} \wedge \circ\neg\mathit{head\_mgn}) \vee (\mathit{new\_ins} \wedge \circ\neg\mathit{new\_ins})$$

The *new_on_feed* macro captures the event that a new blank appears on the feed belt by either being dropped by the crane or being newly inserted into the system. Thus, the formula expresses that once an object is deposited onto the feed belt, there will be no further deposit until sensor 14 (*feed_sen*) signals that a new object arrived at the end of the feed belt.

4(b) A new blank may only be put on the deposit belt, if sensor 13 confirms that the last one has arrived at the end of the deposit belt.

$$\Box(\mathit{new\_on\_deposit} \Rightarrow \\ \circ(\neg\mathit{new\_on\_deposit}\,\mathcal{U}\,(\neg\mathit{deposit\_sen} \wedge \circ\,\mathit{deposit\_sen})))$$

where

$$new\_on\_deposit = (robot\_mgn[1] \wedge \circ\neg robot\_mgn[1])$$

This property is very similar to 4(a), but for the deposit belt.

4(c) Do not put blanks on the table if it is already loaded.

$$\Box(push\_on\_table \Rightarrow \circ(\neg push\_on\_table\ \mathcal{U}\ robot\_pickup))$$

where

$$push\_on\_table = (feed\_sen \wedge \circ\neg feed\_sen)$$
$$robot\_pickup = (\neg robot\_mgn[0] \wedge \circ\, robot\_mgn[0])$$

Once we push a blank on the rotary table, we should not push another one onto it until the first one is picked up by the robot. Pushing a blank on the rotary table means sensor 14 (*feed_sen*) changes its state from true to false. The robot picks up an object from the rotary table when the magnet on arm 1 changes its state from false to true.

4(d) Do not put blanks into the press if it is already loaded.

$$\Box(drop\_in\_press \Rightarrow \circ(\neg drop\_in\_press\ \mathcal{U}\ pickup\_from\_press))$$

where

$$drop\_in\_press = (robot\_mgn[0] \wedge \circ\neg robot\_mgn[0])$$
$$pickup\_from\_press = (\neg robot\_mgn[1] \wedge \circ\, robot\_mgn[1])$$

Once we load the press, it should not be loaded again before it is unloaded. We load the press when the magnet on arm 1 goes from true to false and we unload it when we pick something up with arm 2 (that is, when the magnet on arm 2 changes state from false to true).

4(e) Do not move the loaded robot arm 1 above the loaded table, if the latter is in unloading position.

$$\Box(((robot\_mgn[0] \wedge \neg abovetbl) \Rightarrow \circ\neg(robot\_mgn[0] \wedge abovetbl)) \wedge$$
$$abovetbl \Rightarrow (rotary\_unloading\ \mathcal{U}\ \neg abovetbl))$$

where

$$abovetbl = (robot\_pos = POS\_ARM1\_TABLE \wedge \\ robot\_ext[0] = EXT\_ABOVETABLE)$$

The first conjunct says that if the robot may have a blank at arm 1 (this is the case when *robot_mgn[0]* is true) and arm 1 is not above the rotary table then it will not move above the rotary table, regardless of the position of the table. The second conjunct tells us that the robot will move away from the rotary table before the table moves. Together, these two properties imply that there would be no case in which the rotary table will have a blank in the unloading position and the robot will move above it holding another blank.

B Implementation Details

A relatively clean way of translating the PROMELA code to C is by the use of a multi-threading library [NBF 96]. We will describe below the translation of each PROMELA construct that we used. We made the following simplifying assumptions:

- We assume that read and assignment statements are atomic. Given that the PROMELA types are `bool`, `byte`, `short`, and `int` this is not an unreasonable assumption for most modern machines;
- We do not care about wasted CPU time. Thus, we use busy loops instead of blocking on a semaphore (or other synchronization device).
- Each global variable can be modified by only one `proctype`;
- Channels are synchronous; each channel is only used by just two processes: one writes to it and another reads from it;
- We run only one thread (`proctype`) of any given type.

The following outlines a transformation scheme:

- Declare each `proctype` as C function;
- Take care of small syntactic issues like the fact that PROMELA uses ";" as a command *separator* while in C ";" is a command *terminator*;
- Transform the `init` process in the `main` function;

- Transform every `run` statement into a create thread and a run thread statement;
- Each boolean guard `b` gets translated into `while(!b);`
- Each `if` statement of the form:

```
if
::t1->a
::t2->b
...
::tn->z
fi;
```

gets translated into:

```
tmp1:  if(t1) {a;}
       else if(t2) {b;}
       ...
       else if{tn} {z;}
       else goto tmp1
```

- Each `do` statement of the form:

```
do
::t1->a
::t2->b
...
::tn->z
od;
```

gets translated into:

```
do{
  if(t1) {a;}
  else if(t2) {b;}
  ...
  else if{tn} {z;}
}
```

- Each channel read of the form:

```
in?v1,v2,v3...vn;
```

gets translated into:

```
read_channel(in, &v1, &v2, ..., &vn);
```

The `read_channel` can be implemented along the following lines:

```
void read_channel(chan_t in, int *v1, ...)
{
  while(!in->data_ready);
  for(i=1; i<=n; i++)
    vi=in->data[i];
  in->data_ready=0;
  in->data_taken=1;
}
```

- Each channel write of the form:

```
out!v1,v2,v3...vn;
```

gets translated into:

```
write_channel(out, v1, v2, ..., vn);
```

The `write_channel` can be implemented by:

```
void write_channel(chan_t out, int v1, ...)
{
  for(i=1; i<=n; i++)
    in->data[i]=vi;
  in->data_ready=1;
  while(!in->data_taken);
  in->data_taken=0;
}
```

This translation scheme is not general by any means. It is simplified as much as possible given the specifics of our model. These assumptions, although notably quite strong, make the translation very easy. But this was our goal right from the beginning.

References

[LL 95] Claus Lewerentz, Thomas Lindner (Eds.) *Formal Development of Reactive Systems. Case Study Production Cell.* Springer-Verlag, Berlin, 1995.

[HO 97] G.J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, 279-295, Vol. 23, No. 5, May 1997

[NBF 96] Bradford Nichols, Dick Buttlar, Jackie Farell. *Pthreads Programming.* O'Reilly & Associates, 1996.

[DAC 98] Matthew B. Dwyer, George S. Avrunin and James C. Corbett. *Property Specification Patterns for Finite-state Verification*, 2nd Workshop on Formal Methods in Software Practice, March 1998.

[CA 95] Thierry Cattel. *Process Control Design Using SPIN*, Spin Workshop, Montreal, Canada, 1995.

[CPS 93] R. Cleveland, J. Parrow, B. Steffen. *The Concurrency Workbench: A semantics-based Tool for the Verification of Concurrent Systems.* ACM TOPLAS, Vol. 5, No. 1, January 1993